Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

L.G.L.T. Meertens

Constructing a calculus of programs

Computer Science/Department of Algorithmics & Architecture Report CS-R8914 April

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.G.L.T. Meertens

Constructing a calculus of programs

Computer Science/Department of Algorithmics & Architecture Report CS-R8914 April

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Constructing a Calculus of Programs

Lambert Meertens
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
University of Utrecht, Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

A large part of the effort in formal program developments is expended on repeating the same derivational patterns over and over again. The problem is compounded by notations that require many marks on paper for expressing one elementary concept, and 'administrative overhead', consisting of algorithmically uninteresting but technically necessary steps, like shuffling parts of an expression around without change in computational meaning, and the introduction of local auxiliary definitions for lack of a suitable notation for what is being defined. This can to a large extent be avoided by developing suitable theories, including a notation that is designed to increase the manipulability. After a reflexion on some of the issues, the more technical part of this paper is devoted to an attempt to construct a system of combinators that is better amenable to manipulation than the classical ones.

1987 CR Categories: F.3.1, F.3.3, F.4.1, D.2.1, D.3.3
1985 Math. Subj. Class.: 68N05, 68Q99

Note: To appear in the proceedings of the International Conference on *Mathematics of Program Construction,* LNCS, Springer-Verlag.

Report CS-R8914
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

0. Introduction

Program construction is a mathematical activity. By 'mathematical activity' is not meant: the actual practice of professional mathematicians, but: establishing properties of formal objects with perfect certainty, that is, as a tautology. Often the property to be established is known, but the formal object that has to enjoy the property is only partially constructed. The programmer's task is to complete the construction. This constructive type of problem is not uncommon in general mathematics, but it is the pre-eminent type in programming. The formal objects concerned are expressions in some formal language. This includes both programs and (formal) specifications.

Programs can themselves be viewed as specifications, in two ways. One is the operational viewpoint: programs as specifying a process for some (abstract) machine. The notion of efficiency is intimately tied to this viewpoint: it is meaningless to discuss the efficiency of a program outside the context of a mapping to a process on a machine. Processes have 'observable' aspects (input and output to the outside world) and purely internal aspects, and the agreement is that (apart from efficiency) only the observable aspects count. This makes the notion of program optimisation meaningful.

We can also abstract from the internal process aspects by identifying observably equivalent processes, and consider the meaning of a program as a point in the resulting abstract space. We then obtain the 'declarative' viewpoint of programs as specifications. It is this viewpoint we are concerned with here. The notion of efficiency, although still a major pragmatic concern and a motive force in our design choices, is thereby moved out of the formal arena. The advantage is that we obtain a rich structure of relations between programs.

Consider what is needed for constructing a program: a formal language for expressing the specification as a formal object, a formal language for expressing programs, and rules that can be used to establish properties.

Let us assume that there are no *a priori* constraints on the formalisms. Can the resulting freedom be used to design formalisms that make the task of program construction easier?

1. On the need for powerful theories

No mathematician could do significant work on the basis of pure ZFC, pure predicate calculus, or pure lambda calculus. Instead, the starting point is a body of theories, with definitions, notations and theorems.

Likewise, for significant program development we need powerful theories, with definitions, notations and theorems that allow to capture large chunks of development in a single step.

Here is a simple example. Let f be a function on the naturals, with an inductive definition of the form:

$$f(0) = e \quad ,$$

$$f(n+1) = g(h(n), f(n)) \quad .$$

The value of $f(N)$ can be computed with the following iterative program, in which the result is the final value of the variable a:

```
|[ $a$: **appropriate-type**
; $n$: **nat**
; $n := 0$; $a := e$
; **do** $n \neq N \rightarrow$
      $a := g(h(n), a)$; $n := n+1$
; **od**
]|
```

A simple theorem, that has been rediscovered many times, is that under certain conditions the computation may also be arranged as follows:

```
|[ $a$: **appropriate-type**
; $n$: **nat**
; $n := N$; $a := e$
; **do** $n \neq 0 \rightarrow$
      $n := n-1$; $a := g(a, h(n))$
; **od**
]|
```

There may be good reasons to prefer this computation schema. The conditions under which it applies are that g satisfies the functional equation

$$g(a, g(b, c)) = g(g(a, b), c)$$

and e is such that

$$g(e, a) = g(a, e) = a$$

for all a.

Most introductory programming texts do not mention this simple theorem. Should they? Its proof is straightforward enough, for example using standard techniques from DIJKSTRA & FEIJEN [3] The only aspect that possibly requires some inventiveness is the choice of the invariant, namely

$$g(a, f(n)) = f(N) \quad .$$

It might, therefore, be argued that—since this fact can be derived on the spot when it applies—no theorem is needed here.

If carried to its extreme, this argument denies the value of having theorems at all. For example, it is not hard to derive the fact that

$$\frac{\mathrm{d}}{\mathrm{d}x} x^n = n \cdot x^{n-1}$$

when the need arises by applying standard techniques for computing limits. The strength of the Differential Calculus is of course that it is a *calculus*. It gives a method for computing a certain kind of limits by following a set of rules rather mechanically, using pattern matching and simple equational reasoning. This is only possible by virtue of a suitable notation geared towards these rules.

Coming back now to the simple theorem mentioned above, a possible reason for not formulating it explicitly is the lack of a suitable notation. As presented above, it takes up half a page. What is worse, the interesting part of it, from a calculus-oriented point of view, is not so much that there are two solutions to one problem, but that the two programs are equivalent under the given condition. It is not difficult to imagine that in a less abstract problem setting it would be hard to see by pattern matching that the theorem applies.

A formalism whose aim is to provide the kind of notation by which this and similar theorems can be formulated concisely, so that they can be applied as part of a calculus, was developed in [4, 1, 2]. The condition on g and e is precisely that they are the operation and identity element of a monoid. It is more pleasing then to denote g as an infix operator, say $\oplus$, which by the monoid properties is associative. The function f takes a natural as argument, but a slightly more general and abstract viewpoint is that the computation has the sequence

$$[h(0), h(1), \ldots, h(N-1)]$$

as its argument, and therefore *any* list. The function applied to this list 'reduces' it to a single value by combining the elements using the operation $\oplus$. A short notation for this function is

$$\oplus/ \quad ,$$

which is borrowed from APL but expresses in addition, when applied to a sequence, that its operation is that of a monoid, and thereby that there are many different orders in which the reduction can be computed.

For the two program schemas above, one in which the reduction is performed 'from left to right', and one in which the computation proceeds 'from right to left' (or in any case in the opposite direction), the operation is not required *per se* to be associative. It might even be the case that the two operands have different types, whereas they are of the same type for an associative operator. Concise notations for these two computation schemas are

$$\oplus\not\rightarrow_e \quad \text{and} \quad \oplus\not\leftarrow_e \quad .$$

The theorem, formulated with the aid of this notation, is now:

$$\text{If } (\oplus, e) \text{ is a monoid, then } \oplus\not\rightarrow_e \;=\; \oplus\not\leftarrow_e \quad .$$

The reader familiar with [1] or [2] will recognise this as (a mildly specialised version of) the Specialisation Lemma. The point is that in this form the theorem has the right characteristics to being useful as ingredient in a calculus.

It is not hard to understand why having theories is important. Given a significant piece of mathematical work, it is (theoretically) possible to make it entirely self-contained, theory-free so to say, by including definitions of all notations used, and statements and proofs of theorems invoked, and again definitions and proofs of notations and theorems used in there, down to some basic level. Theorems and notations that are used only occasionally, perhaps once, can be expanded in place. The result will not be pleasing. But this is how the work would look if we did not have theories to work from at our disposal. The prevailing situation in (formal) program construction is, unfortunately, not much better.

There is, therefore, reason for the hope that the currently often excessive length of rigorous formal program developments is not so much due to the need for rigour, but rather, at least to a large extent, to the lack of a suitable body of theories to build upon. The experience with our, thus far modest, formalism provides some evidence for this.

2. Algorithmics anonymous

What now, precisely, are the characteristics that make this formalism suitable for doing program construction by calculation? One is that it is indeed modest. As the formalism is developed over time, more notation is added, and there is the constant need to be extremely careful here. Unchecked, it would explode into a flurry of special symbols.

The existing notations have been developed with an eye to conciseness of expression, taking account of what is actually often encountered. Conciseness is important for making the recording of the development steps less laborious, and also for making it possible in the first place to do the pattern matching needed to recognise the applicability of some rule.

A further advantage is that the need is diminished to *interrupt* the smooth, linear, development for doing an 'aside' sub-development. It is quite normal in a development to single out a subexpression of the main expression, derive an equivalent form for it, and substitute the result back. Whether and when this is done, and if so how, is a design choice in the presentation of the argument. The subdevelopment may precede the main development, perhaps as a lemma, or it may be a 'deferred justification'. In principle, it could be expanded in place. A good reason for not doing so may be the structuring of the argument. Especially if the context surrounding the subexpression is large, this device may increase readability by zooming in on the symbols where the action is. If, however, that context is mainly large because of the verbose notation, the interruption is more a matter of practical necessity than of choice.

Given a concise notation, the choice between singling-out versus having an uninterrupted, linear derivation, is up to the designer of the presentation, and it is a good thing to have this freedom. However, there are also cases where a separate subdevelopment is forced for purely technical reasons. This can be an annoying interruption of the argument. A further goal, therefore, of the formalism is to avoid this.

One important case where this phenomenon pops up is if recursively defined names are used. By 'name' here a single symbol is meant that refers to a definition elsewhere. In that sense, the expression '$2+2$', although denoting 4, is not a name for it. If a name has a recursive definition, this means that it is not possible to just replace the name by its definiens. The latter contains further occurrences of that name (by the definition of 'recursive definition'), and so the expression after substitution must—in contrast to what happens for non-recursive definitions—still be interpreted in the context of the original definition. Usually the well-foundedness of the recursion corresponds to a split in the definition into a base case and other cases, and this may generate then the need for a case analysis, another technical reason for an interruption. And, finally, recursive definitions often give rise to the need of a proof by induction, which cannot be done 'in place'.

The counterpart to this is if the *solution* to which a development leads requires, in the language used, a recursive definition, and thereby a name for the recursively defined part. Without some rather special mechanism, this also necessitates an aside in the development.

A theoretical solution is the use of a fixpoint combinator, but the 'theoretical' should be emphasised here; such a combinator is not particularly pleasant if it comes to calculating with it. The approach adopted in our formalism is to provide explicit notations for the solutions to the most frequent recursive definition patterns, and to give a set of laws to go with it. Above, we have seen the notation $\oplus/$; this is one such notation. The subexpression that before required a name, now can remain *anonymous*. This extremely simple stratagem buys us a good deal of calculational manipulability. It should, in fact, be familiar to every programmer. The 'while loop'

$$\textbf{while}\ p\ \textbf{do}\ S\ \textbf{od}$$

is an explicit anonymous notation for the solution to this recursive definition for W:

$$W \;=\; \textbf{if}\ p\ \textbf{then}\ S\,;\ W\ \textbf{fi} \quad .$$

3. Further causes of laboriousness

Names are not by themselves bad, of course; attempts directed towards totally abandoning names lead to a tar pit. It is being forced to pick and use ephemeral names for items that do not correspond to any abstraction worth naming, that is the cause of much additional labour. After 'recursive' names, the next target is formed by dummy (bound) variables, as in function definitions or lambda forms. The point here is that a variable-free definition is at least potentially more manipulable. The extra notation needed for denoting the dummy variables, and for delimiting the scope, tends to get in the way of the easy manipulation. In our little formalism one contribution here are the so-called sections; instead of writing something like

$$\lambda x : a \oplus x \quad ,$$

in which one operand is fixed, leaving a monadic function, the pithy notation

$$a\oplus$$

can be used. This saves us many, many marks on paper. The alternative in such cases is hardly to use the lambda notation. Presently, if the existing devices for getting rid of dummy variables do not suffice, the best alternative is to interrupt the development for a definition, like: "Putting

$$f x \;=\; a \oplus x \quad ,$$

we have ..." and so on. Again, introducing a name here is not necessarily bad; the bad thing is being forced to it. One form of this that is all over the existing papers using the current formalism occurs when an operator is needed, like in: "Putting

$$a \odot b \;=\; a \oplus (f b) \quad ,$$

we have ..." and so on. The interruption is forced in this case by the lack of an explicit, closed-form expression for the solution $\odot$ of the functional identity $a \odot b \;=\; a \oplus (f b)$.

What we need here are combinators that allow expressing that solution in terms of components like $\oplus$ and f. The 'classical' combinators **S** and **K** will make anything variable-free, but they do not have the desirable manipulative properties. If it is sometimes nice that 'there is only one thing you can do', these combinators virtually force us on a single development track: they act only at the head of a tree, and most of the steps are just shuffling to get things up there. There is also the other direction, in which combinators are not expanded but introduced, and here there are usually too many possibilities, and no heuristics for choosing among these. Worst of all, in proving the equivalence of two combinator expressions, the standard technique requires introducing dummy variables for unsatiated combinators.

The basic problem here is that the basic operation of the classical combinator calculus (and also of the closely related lambda calculus) is application instead of composition. Application has not a single property. Function composition is associative and has an identity element (if one believes in the 'generic' identity function).

Often seemingly minor notational issues make a huge difference. A well known example is the use of infix notation for associative operators. If much use of the associative property is made (and for function composition it is), the just sufficiently ambiguous infix notation

$$f \circ g \circ h$$

saves one calculation step for each case.

In [4] a notational suggestion was made, not followed in [1] and [2], for a further ambiguity-on-purpose. It is the device called 'apposition' there, to denote function *app*lication and com*position* in the same way. This saves us the trivial step in

$$f(g(x)) \;=\; (f \circ g)(x) \quad .$$

There is a problematic aspect to this notational trick (dubbed, with another portmanteau, 'complication' by Bird): it requires knowing the types of the constituents to parse the train f-apposed-to-g-apposed-to-x. In a polymorphic context, it is in general impossible to guarantee that x will not be substantiated with a function, which would radically alter the meaning. So apposition is not so substitutive as is desirable.

4. Whither, apposition?

Apposition may perhaps be madness, but there is some curious (Dutch?) method to it. A close relative is the silent 'lifting' of operators to functions, another notational device in [4] that has not met with universal acclaim, whereby each operator $\oplus$ could be overloaded to also denote the operator $\hat{\oplus}$ such that

$$f \hat{\oplus} g \;=\; \lambda x\colon (f\,x) \oplus (g\,x) \quad .$$

The relationship can be seen if we consider an operation that does not use its left operand. If the lifting is not denoted, the definition reads then: $\oplus$ applied to g is $\oplus$ composed with g. This suffers from the same problems as apposition does. The most irritating thing here is that these problems are real, but encountered rarely in practice (at least until now). The extra steps needed on giving this up are also real, and very frequent. One theoretically sound way of saving this device is to agree that also constants in the formalism are silently lifted to constant functions, so that '2', for example, is the name of the function

$$\lambda x\colon \text{“the successor of the successor of zero”} \quad .$$

As the circumlocution in the body shows, we have lost the name for the number itself. This is not appealing; who wants to live with spurious identities like $2 \circ 3 = 2$?

There is something in common to many of the (usually minor) annoying problems in the use of the formalism. They all point in the direction of the need of a suitable system of combinators for making functions out of component functions without introducing extra names in the process. Composition should be the major method, and not application. Among the further desirable properties is that the system should not be opposed to a typing discipline. (The classical combinators do not permit any reasonable form of typing.) c Also, less effort should be needed for 'administrative' steps that serve to bring the 'data' in the expression to the spot where the action is (typically a substantial part of the work). In particular, the choice between the asymmetric 'Curried' view on the type of a function with two arguments, say

$$\alpha \rightarrow (\beta \rightarrow \gamma)$$

and the flat view

$$(\alpha \times \beta) \rightarrow \gamma$$

should be reasonably light-weight, and there should be no built-in bias for the first of the two for operators. Finally, it is desirable that functions can as easily and gracefully deliver a tuple as result as they will take it as an argument, facilitating composition.

The remainder of this paper is devoted to an attempt to construct such a set of combinators. The starting point is a type system that centres on functions taking (typed) tuples as arguments and giving tuples as result. The notion of combinator is next taken rather literally; it is examined how such functions can be combined, more or less as if they were wheeled in as physical boxes with output lines that can be connected to the input lines of other boxes.

5. A TYPE SYSTEM

We start with a typed universe of 'plain values', not containing tuples or functions. From the plain types we construct 'singleton types', 'tuple types' and 'function types', in a mutually recursive fashion. Greek letters, possibly adorned with subscripts, will serve as variables that stand for types. Specifically, σ, τ, υ and ω will be used for tuple types, and α, β and γ for singleton types.

Each function type is formed from an *out-type* (for the codomain) and an *in-type* (for the domain), both of which are tuple types. If σ and τ are two tuple types, then

$$\sigma \leftarrow \tau$$

denotes the function type with out-type σ and in-type τ. A more conventional notation would be $\tau \rightarrow \sigma$.

A tuple type is formed from a finite sequence of zero or more singleton types. Its *width* is defined to be the length of that sequence. Let a sequence be given of n singleton types α_i, $0 \leqslant i < n$. The corresponding tuple type has then width n. If $0 < n$, it is denoted by

$$\alpha_0 , \alpha_1 , \cdots , \alpha_{n-1} \quad .$$

This resembles the n-ary operation $\_\times\_\times\cdots\times\_$ of the Cartesian product, but here the operation $,$ is considered to be 2-ary and associative. For example,

$$\alpha,\beta,\gamma \quad ,$$

$$(\alpha,\beta),\gamma \quad ,$$

and

$$\alpha,(\beta,\gamma)$$

all denote the same tuple type. So if σ and τ are tuple types, then so is σ,τ. Its width is the sum of the widths of σ and τ. On purpose the notation does not distinguish between singleton types and tuple types of width 1. These are identified. We need a special notation for the (unique) tuple type of width 0, for which the symbol $\mathbb{1}$ will be used. Under the operation $,$, the tuple types form a monoid, with identity element $\mathbb{1}$ (so $\sigma,\mathbb{1}$ and $\mathbb{1},\sigma$ both denote the same type as σ).

Finally, the singleton types consist of the plain types, together with the function types.

Here is a BNF grammar for the 'type expressions' as described above, assuming a predefined metasyntactic variable ⟨plain type⟩:

$$\langle\text{function type}\rangle ::= \ (\langle\text{tuple type}\rangle \leftarrow \langle\text{tuple type}\rangle)$$

$$\langle\text{tuple type}\rangle ::= \ \langle\text{tuple type}\rangle,\langle\text{tuple type}\rangle \mid \langle\text{singleton type}\rangle \mid \mathbb{1}$$

$$\langle\text{singleton type}\rangle ::= \ \langle\text{plain type}\rangle \mid \langle\text{function type}\rangle$$

What this grammar does not express, of course, is that $,$ is associative and has identity $\mathbb{1}$.

The parentheses in the first syntax rule are needed to distinguish between, e.g., the function types

$$(\sigma \leftarrow (\tau \leftarrow \upsilon))$$

and

$$((\sigma \leftarrow \tau) \leftarrow \upsilon) \quad .$$

If no ambiguity can arise, these parentheses may be dropped. They also serve to distinguish between, e.g., the tuple type

$$\alpha,(\sigma \leftarrow \tau),\beta$$

and the function type

$$\alpha,\sigma \leftarrow \tau,\beta \quad ,$$

which is interpreted as $(\alpha,\sigma) \leftarrow (\tau,\beta)$.

The various new kinds of types are inhabited by values, exactly in the way that would be expected. So a value of type α,β,γ, for example, is a 3-tuple consisting of an α-, a β- and a γ-value. There is exactly one, not very interesting, value of type $\mathbb{1}$. A function of type $\sigma \leftarrow \tau$ yields a σ-tuple when provided with a τ-tuple as argument. The width of σ is called the *out-width* of the function, and the width of τ its *in-width*. It is not assumed that functions are total.

The type system can be made polymorphic in a way that has become usual, with, for example, a generic identity function 'id' of the *polymorphic* type $\alpha \leftarrow \alpha$. However, such type polymorphism will only be exercised with the constraint that refinement of polymorphic types preserves the widths involved. As we shall see, it is essential that the out- and in-widths of functions denoted by expressions can be determined from those of the constituents, and the superposition of type polymorphism must not break this. We use the notation

$$\vec{\alpha}^n \quad = \quad \alpha_0, \alpha_1, \ldots, \alpha_{n-1}$$

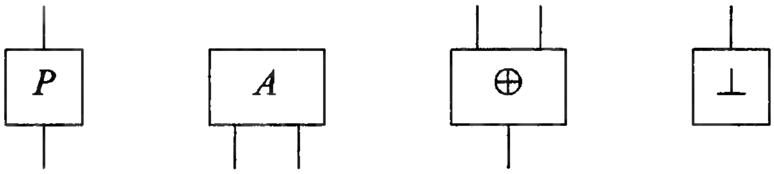
for the unrestrained polymorphic type of width n. For each width n there is a different generic identity function

$$\mathrm{id}_n \colon \vec{\alpha}^n \leftarrow \vec{\alpha}^n \quad ,$$

and id_1 is usually abbreviated to just id.

6. The function world

From now on we are only interested in functions. Here are some schematic pictures of functions:



The functions are depicted as boxes. The argument tuple is fed into a box through the lines at the top, and the result tuple appears at the bottom. By convention, the flow in these schemas is always from high to low, so that there is no need to put arrowheads on the lines. The boxes are labelled with names for the functions. Let us consider them one by one.

The function P takes a singleton argument and delivers a singleton result.

The function A has in-width 0. Its argument is the uninteresting 0-tuple, which is the only inhabitant of the type $\mathbb{1}$. Since it carries no information, there is no need to show the 0 incoming lines more vividly. Functions with in-width 0 are called *sources*.

A source, like this function A, can be thought of as modelling a (constant) value in the function world. The result of A is a 2-tuple, or pair, of some type α, β. The type of A itself is then $\alpha, \beta \leftarrow \mathbb{1}$. The names of sources, that is, functions with in-type $\mathbb{1}$, will in general be taken from the initial part of the alphabet.

The function $\oplus$ has out-width 1 and in-width 2. As the example shows, the names of functions may be special symbols. Some more examples of symbol names are $+\!\!+$, $\odot$, $*$ and $/$. Functions with symbol names are also called 'operators'. Some of these symbols, like $+\!\!+$, denote by convention a specific function. For example, $+\!\!+$ stands for sequence concatenation; it has type $[\sigma] \leftarrow [\sigma], [\sigma]$. Others, like $\oplus$ and $\odot$, have no fixed meaning. They are true variables, just like P and A. The ordering of the incoming lines is significant. If the type of $\oplus$ is $\sigma \leftarrow \alpha, \beta$, the α-value is supposed to be carried by the left in-line, and the β-value by the right in-line. Similarly, the out-lines of a box carry from left to right in order the singleton values of which the output tuple is composed.

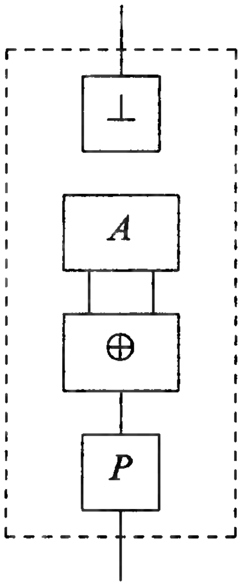
The last function can be called a terminator, or *sink*. It accepts a value but delivers no information. In general, a sink is any function whose out-type is $\mathbb{1}$. A sink is (up to polymorphic type refinement) fully determined by its in-width. The name of the unique sink of type $\mathbb{1} \leftarrow \alpha$, $\perp$, is thus chosen because of its graphical representation of the function as terminator. It bears no relationship to the bottom of a lattice. For the sink of in-width n we shall use the name $\perp_n$.

Not depicted above is $\perp_0$, the sink of in-width 0, the only sink that is also a source. This is the dullest function imaginable. It has some marginal theoretical interest though. Another name for this function is id_0.

7. Serial composition

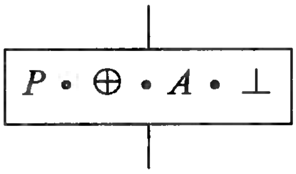
We want to have a set of *combinators* that form functions from given functions. For the purpose of combining functions, they are black boxes. A combinator cannot 'inspect' a function. From a mathematical point of view, a combinator is nothing but a higher-order function, but here these combinators are emphatically not considered to live in our function world. The most important combinator is *serial composition*. We shall also encounter parallel composition. If used without qualification, plain 'composition' will mean: serial composition.

Below we see the serial composition of a few boxes.



This composite will be expressed as $P \bullet \oplus \bullet A \bullet \perp$, in which $\bullet$ can be pronounced as 'dot'.

The purpose of the dashed box is to suggest that we can abstract from the details of the composition, and treat this as one new box:



For the composite to be meaningful, the out-type of each box has to be compatible with the in-type of the box it interfaces to. In a polymorphic setting this may entail refining these types. It is understood then that the unifying type substitutions are consistently performed throughout the type of a box. In the schematic pictures it is always assumed that the components have types for which the whole combination is meaningful.

If P has out-width m and in-width n, then we have

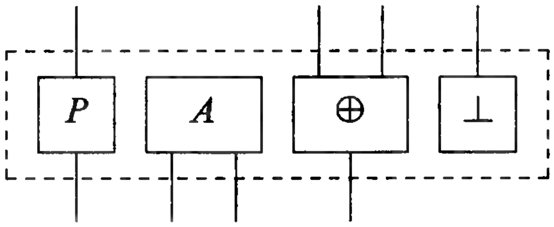
$$P \;=\; \mathrm{id}_m \bullet P \;=\; P \bullet \mathrm{id}_n \quad .$$

In the schematic diagrams, a function id_n will not be shown explicitly as a box, but corresponds to any group of n adjacent collateral lines.

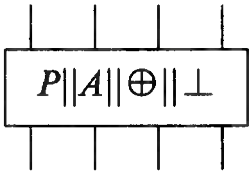
Thus far, it was tacitly assumed that the widths at each interface are equal. Such *balanced* serial composition is just the usual function composition. However, we will move on to a generalisation in which the widths do not have to balance. But first another form of composition is introduced.

8. Parallel composition

The following picture depicts parallel composition:



This parallel composite will be expressed as $P||A||\oplus||\perp$, where $||$ is pronounced 'para'. We may abstract as before:



The out-width of a parallel composite is the sum of the out-widths of the components, and likewise for the in-width.

Like serial composition, parallel composition is associative. There is also an identity element of $||$, namely id_0. Two further simple laws relating identity functions and sinks to parallel composition are:

$$\mathrm{id}_{m+n} = \mathrm{id}_m||\mathrm{id}_n \quad ,$$

and

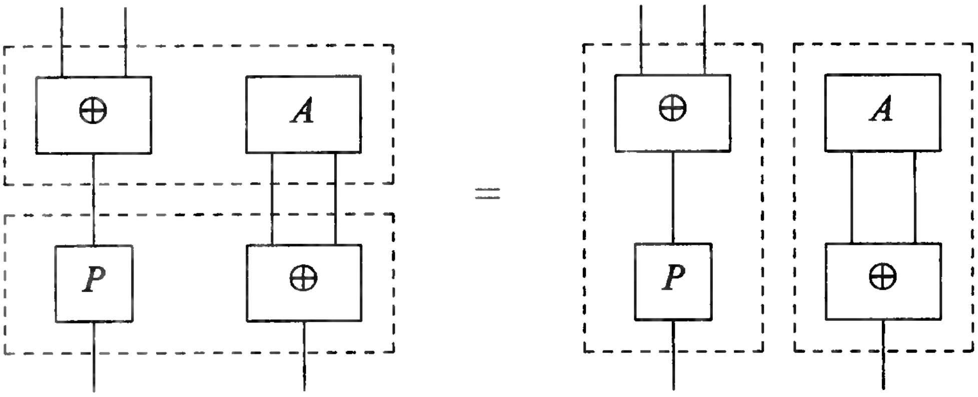
$$\perp_{m+n} = \perp_m||\perp_n \quad .$$

We now come to a law relating serial and parallel composition. In a mixed expression, involving both the combinator || and the combinator • , the first will take precedence. In fact, serial composition has the *lowest* priority of all combinators that will be introduced here. For the others no relative priorities are defined, and so parentheses will be needed to specify grouping in mixed expressions.

Now the promised law. If the compositions involved are balanced, the following law holds:

$$f||F \bullet g||G = (f \bullet g)||(F \bullet G) \quad .$$

(The requirement that the two compositions of the r.h.s are balanced is sufficient to guarantee balanced composition in the l.h.s.) In the terminology of [2], || *abides* with • . The abide property is illustrated in the picture below:



This law is not particularly important, but it serves to pose the question: What exactly do we mean by equality of two box expressions, built with (box) combinators? This could be defined in terms of a semantic domain involving functions, but a much simpler answer is possible: *Two box expressions are equal if the combinators specify networks with identical* topologies *between the component boxes.* This generates an algebra of boxes and combinators. If for certain boxes further properties are specified, for example, 'P is idempotent' (that is, $P \bullet P = P$), we can take the free algebra *modulo* these properties.

The box algebra is of course what we are interested in. There is some didactic advantage in the fact that various properties can be illustrated by diagrams, but proving complicated properties by pictures becomes laborious.

9. Serial composition revisited

We shall now extend the definition of serial composition so as to allow *unbalanced* composition of boxes. This will be done by reducing it to balanced composition. The idea is the following: if the interfacing widths in a serial composition differ, the box with the deficit is 'stretched' to the required width by extending it to the right, by means of parallel composition, with an identity function whose width makes up the deficit. This can be defined more formally. Let the operation $\dot{-}$ between two naturals be defined by

$$m \dot{-} n \quad = \quad (m \uparrow n) - n \quad ,$$

in which $\uparrow$ denotes the operation of taking the maximum of the two operands, and $-$ is conventional subtraction. It is immediate that

$$(m \dot{-} n) + n \; = \; (n \dot{-} m) + m \quad .$$

Also,

$$(m \dot{-} n) \downarrow (n \dot{-} m) \; = \; 0 \quad .$$

Let now P and Q be two functions, and let m be the in-width of P and n the out-width of Q. Then we define

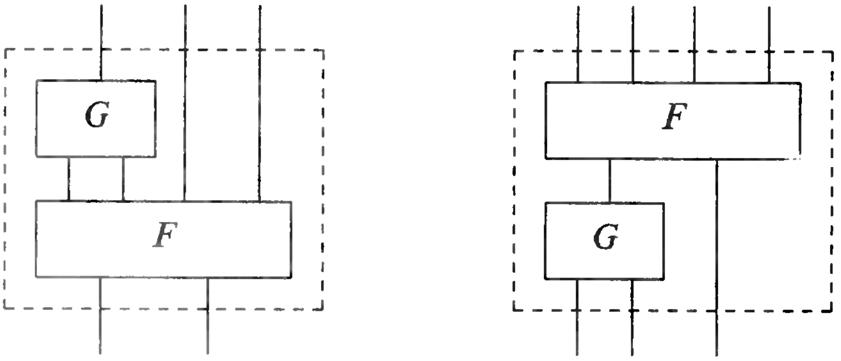
$$P \bullet Q \quad = \quad (P || \mathrm{id}_{n \dot{-} m}) \bullet (Q || \mathrm{id}_{m \dot{-} n}) \quad .$$

The parallel composition with identity functions of the appropriate widths balances the composition in the r.h.s. If we apply this to a composition that was already balanced, the r.h.s. reduces to

$$(P || \mathrm{id}_0) \bullet (Q || \mathrm{id}_0) \quad ,$$

which, since id_0 is the identity of $||$, gives us back the l.h.s.. So the new definition extends the original meaning in a uniform way.

The diagram below shows this in action for each of the two ways in which a composition can be unbalanced. Here F has out-width 2 and in-width 4, whereas G has out-width 2 and in-width 1.



Note that the composite to the left resembles so-called 'partial parametrisation', especially if the box G is replaced by a source. In particular, it is the case that

$$F \bullet (A || B) \quad = \quad (F \bullet A) \bullet B \quad ,$$

provided at least that A is a source.

The important property making this more general form of serial composition useful is that this combinator is still associative. The composition combinator has an identity element, namely id_0, which was also the identity of parallel composition. (Note that $\mathrm{id} = \mathrm{id}_1$ is not an identity element, unless we restrict the domain of $\bullet$ to functions for which neither the in- nor the out-type is $\mathbb{1}$.)

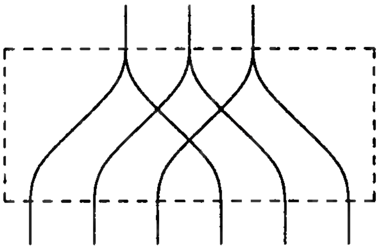
There is a price to be paid for the additional flexibility afforded by the generalisation. It is the obligation to keep track of the widths. Quite a few of the laws involving composition have conditions on the widths concerned, for example of the pattern: 'provided that the in-width of P is at least the out-width of Q'. There is an opportunity for human errors here if the (usually boring) verification of such conditions is unduly omitted. In a mechanical system for providing assistance to reasoning with equalities of box expressions, the verification could be delegated to general type checking.

10. Input sharing

The next combinator will allow several boxes to share their inputs. As a purely auxiliary group of generic functions we introduce first, for each natural n,

$$\Lambda_n\colon\ \vec{\alpha}^n , \vec{\alpha}^n \ \leftarrow\ \vec{\alpha}^n \quad ,$$

which produces a $2n$-tuple from an n-tuple by joining two copies together. The following diagram shows Λ_3:



As before, we first define a balanced form of 'sharing', and extend the definition next to the general case. Let P and Q be functions with compatible in-types, which implies in particular that they have the same in-width. Let n be that in-width. Then we define

$$P,Q \;=\; (P\|Q) \bullet \Lambda_n \quad .$$

In words, the input to P,Q is fed to both P and Q, and the output of P,Q is then obtained by joining the two resulting output tuples. Both $(\mathrm{id}\|\bot)$ and $(\bot\|\mathrm{id})$ have out-width 1 and in-width 2, so

$$(\mathrm{id}\|\bot),(\bot\|\mathrm{id})$$

has out-width 2 and in-width 2. It is easily seen (by 'proof by picture') to be id_2. If the two parallel composites are switched, thus:

$$(\perp\|\mathrm{id}),(\mathrm{id}\|\perp) \quad ,$$

we find again out-width 2 and in-width 2. This function switches the two components of a pair. It has polymorphic type

$$\beta,\alpha \leftarrow \alpha,\beta \quad .$$

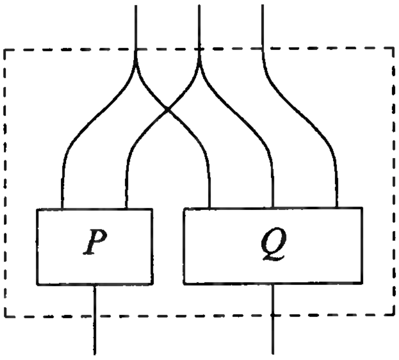
For the general case, in which the components do not have to have the same in-width, let m be the in-width of P and n that of Q. The requirement on the in-types becomes now that $P\|\mathrm{id}_{n \dot{-} m}$ and $Q\|\mathrm{id}_{m \dot{-} n}$ have compatible in-types. This amounts to compatibility of the first $m \downarrow n$ components of the two in-types. For serial composition, balance was achieved by stretching. Here we trim instead the input for the less demanding box to the required length. The function

$$P\|\perp_{n \dot{-} m}$$

has the same out-width as P, but in-width $m \uparrow n$. We define now for the case that $m \neq n$:

$$P,Q \;=\; (P\|\perp_{n \dot{-} m}),(Q\|\perp_{m \dot{-} n}) \quad .$$

As for the earlier combinators, , is associative and has identity id_0. The effect of , is illustrated by:



The line forked off to the left from the third in-line at the top is not shown; it would run into a $\perp$.

11. Type inference rules

The following rules show how the type of a box expression built with $\bullet$, $\|$ and , can be deduced from the types of its components.

$$\frac{\begin{array}{ll} P: & \sigma \leftarrow \tau,\omega \\ Q: & \tau \leftarrow \upsilon \end{array}}{P \bullet Q : \sigma \leftarrow \upsilon,\omega} \qquad\qquad \frac{\begin{array}{ll} P: & \sigma \leftarrow \tau \\ Q: & \tau,\omega \leftarrow \upsilon \end{array}}{P \bullet Q : \sigma,\omega \leftarrow \upsilon}$$

$$\begin{array}{ll} P: & \sigma_0 \leftarrow \tau_0 \\ Q: & \sigma_1 \leftarrow \tau_1 \\ \hline P \| Q : \sigma_0 , \sigma_1 \leftarrow \tau_0 , \tau_1 & \end{array}$$

$$\begin{array}{ll} P: & \sigma_0 \leftarrow \tau , \omega \\ Q: & \sigma_1 \leftarrow \tau \\ \hline P , Q : \sigma_0 , \sigma_1 \leftarrow \tau , \omega & \end{array} \qquad \begin{array}{ll} P: & \sigma_0 \leftarrow \tau \\ Q: & \sigma_1 \leftarrow \tau , \omega \\ \hline P , Q : \sigma_0 , \sigma_1 \leftarrow \tau , \omega & \end{array}$$

For $\bullet$ and $,$ there are two rules, corresponding to the two ways of possible unbalance. For example, if we have

$$A: \sigma \leftarrow \mathbb{1}$$

$$P: \tau \leftarrow \upsilon$$

we can use the second rule for $\bullet$, with $(P, Q) := (A, P)$ and $(\sigma, \tau, \omega, \upsilon) := (\sigma, \mathbb{1}, \tau, \upsilon)$, to deduce

$$A \bullet P : \sigma , \tau \leftarrow \upsilon \quad .$$

If we compute the types of $A \| P$ and A , P, we find in both cases the same type.

It is possible to merge each of the rule pairs by introducing a (partial) tuple 'subtraction' operation, akin to $\dot{-}$ on naturals. By building up a small theory for this operation it becomes possible to prove, for example, that the expressions $(P \bullet Q) \bullet R$ and $P \bullet (Q \bullet R)$ (if any of the two is typable) have the same type without the extensive case analysis that would otherwise be required.

12. Laws

We sum up here, without proof, some important laws for the combinators $\bullet$, $\|$ and $,$. We need a convenient way to denote conditions on the laws.

The notation $P \preccurlyeq Q$ means: the in-width of P is at most the out-width of Q. Similarly, $P \succcurlyeq Q$ means that the in-width of P is at least the out-width of Q. Equality of these widths is denoted as $P \asymp Q$. The composition in $P \bullet Q$ is balanced if and only if $P \asymp Q$. Note that these relations are not transitive, and that $\asymp$ is not reflexive. However,

$$(P \preccurlyeq Q) \wedge (P \succcurlyeq Q)$$

equivales

$$P \asymp Q \quad .$$

If A occurs, it stands for a source. The letter denotes (implicitly) the condition $A \asymp \mathrm{id}_0$.

(L0) $\bullet$, $\|$ and $,$ are associative, with identity element id_0

(L1) $\mathrm{id}_{m+n} \;=\; \mathrm{id}_m \| \mathrm{id}_n$

(L2) If $\mathrm{id}_m \preccurlyeq P$, then $\mathrm{id}_m \bullet P \;=\; P$

(L3) If $P \succcurlyeq \mathrm{id}_n$, then $P \bullet \mathrm{id}_n \;=\; P$

(L4) $\perp_{m+n} \;=\; \perp_m \| \perp_n$

(L5) If $\mathrm{id}_0 \bowtie P \bowtie \mathrm{id}_n$, then $P \;=\; \perp_n$

(L6) $A \bullet P \;=\; A \| P \;=\; A, P$

(L7) $P \| A \;=\; P, A$

(L8) $P \bullet \perp_n \;=\; \perp_n \| P$

(L9) If $p \bowtie q$ and $P \bowtie Q$, then $(p \bullet q) \| (P \bullet Q) \;=\; p \| P \bullet q \| Q$

(L10) If $P \bowtie Q$, then $P \bullet Q \| R \;=\; (P \bullet Q) \| R \;=\; P \| R \bullet Q$

(L11) If $\mathrm{id}_m \bowtie P$, then $P \| Q \;=\; \mathrm{id}_m \| Q \bullet P$

(L12) If $P \succcurlyeq \mathrm{id}_n$, then $\perp_n, P = P, \perp_n \;=\; P$

(L13) If $P \preccurlyeq Q$, then $(P \bullet Q), R \;=\; P \bullet Q, R$

(L14) If in-width $P \leqslant$ in-width Q, then $P, (Q \| R) \;=\; (P, Q) \| R$

(L15) If $P \bowtie R$ and $Q \bowtie R$, then $P, Q \bullet R \;=\; (P \bullet R), (Q \bullet R)$

These laws do not constitute a carefully selected set; it would be interesting to create a nice (independent and complete) set of basic laws. The criterion for inclusion above was more practically inspired. For example, (L11) is essentially a variant of (L10). It is included because it is often applicable in this form, and saves us then one rather trivial step.

13. Formulae and sections.

We introduce some special notations concerning operators, that is, functions denoted by special symbols (non-tags). Throughout this section the variable $\oplus$ stands for an operator with in-width $m+n$, and P and Q for functions with, respectively, out-width m and n. Furthermore, A and B denote sources, also with, respectively, out-width m and n. We define:

$$P \oplus Q \;=\; \oplus \bullet P, Q$$

$$P \oplus \;=\; \oplus \bullet P \| \mathrm{id}_n$$

$$\oplus Q \;=\; \oplus \bullet \mathrm{id}_m \| Q$$

All the compositions involved are balanced. The first form, in which two 'operands' are supplied, is called a 'formula'. The next two forms are a 'left section' and a 'right section'. If the expressions for the operands P or Q are not simple tags, they must be enclosed in parentheses to prevent ambiguities. This is also needed if they are themselves operators: does $\oplus \otimes$ mean $\otimes \bullet (\oplus \| \mathrm{id})$ or $\oplus \bullet (\mathrm{id} \| \otimes)$? The first meaning can be specified by $(\oplus)\otimes$, and the second by $\oplus(\otimes)$. However, if $\oplus$ is an associative operator (see below), then the meaning of $(\oplus)\oplus$, and $\oplus(\oplus)$ is the same (and, as follows from (L16) below, equal to $\oplus \bullet \oplus$), and it is harmless then to drop the parentheses.

These special notations are useful for various reasons. In the first place, algebraic properties from the 'value world' are inherited in the function world. For example, in the usual definition of $\oplus$: $\sigma \leftarrow \sigma,\sigma$ being associative, namely

$$a\oplus(b\oplus c) = (a\oplus b)\oplus c \quad ,$$

the operands are assumed to range over all values of the type σ. Now this is equivalent to the property that

$$F\oplus(G\oplus H) = (F\oplus G)\oplus H \quad ,$$

where this time the operands range over all *functions* with out-type σ. Symmetry (commutativity) and idempotence are likewise inherited.

A second reason for having these notations is that they encode (implicitly) information about the widths. This makes it possible to formulate laws that can be applied without having to verify width conditions separately.

Here are some laws for formulae and sections with a source as operand.

(L16) $A\oplus \;=\; \oplus \bullet A$

(L17) $A\oplus Q \;=\; A\oplus \bullet Q$

(L18) $P\oplus B \;=\; \oplus B \bullet P$

We show how these laws can be derived. For (L16):

$A\oplus$

$=$ { definition of left section }

$\oplus \bullet A\|\mathrm{id}_n$

$=$ { (L6) }

$\oplus \bullet A \bullet \mathrm{id}_n$

$=$ { (L0), (L2) }

$\oplus \bullet A$.

Now (L17) is easily derived:

$A\oplus Q$

$=$ { definition of formula }

$\oplus \bullet A,Q$

$=$ { (L6) }

$\oplus \bullet A \bullet Q$

$$\begin{array}{ll} = & \{\,(\mathrm{L16})\,\} \\ & A\oplus \bullet Q \quad . \end{array}$$

For (L18) we have:

$$\begin{array}{ll} & P\oplus B \\ = & \{\,\text{definition of formula}\,\} \\ & \oplus \bullet P,B \\ = & \{\,(\mathrm{L7})\,\} \\ & \oplus \bullet P\|B \\ = & \{\,(\mathrm{L11})\,\} \\ & \oplus \bullet \mathrm{id}_m\|B \bullet P \\ = & \{\,\text{definition of right section}\,\} \\ & \oplus B \bullet P \quad . \end{array}$$

It is often desirable to have the mirrored version of an operator also available as an operator. Here we encounter a (not entirely unexpected) weakness of the tuple view adopted. What we want, basically, given an operator

$$\oplus\colon \sigma \leftarrow \tau,\upsilon \quad ,$$

is to define a mirrored version

$$\widetilde{\oplus}\colon \sigma \leftarrow \upsilon,\tau \quad ,$$

which is also an operator. However, the type former , leaves no seam when joining two tuple types, so it is not possible to define once and for all, for all operators, what the meaning of mirroring is. In many cases, however, the desired split in τ,υ is clear enough; often it will be in the middle, either because the in-type is of the form α,β, with width 2, or of the form σ,σ. In any case, we assume here that we know how the in-type of $\oplus$ should be split, namely into τ and υ, and that the width of τ is m and that of υ is n. Then, by definition,

$$\widetilde{\oplus} \;=\; (\bot_n\|\mathrm{id}_m)\oplus(\mathrm{id}_n\|\bot_m) \quad .$$

We have:

$$\begin{array}{rcl} Q\widetilde{\oplus}P & = & P\oplus Q \quad , \\ \widetilde{\oplus}P & = & P\oplus \quad , \\ Q\widetilde{\oplus} & = & \oplus Q \quad . \end{array}$$

14. Locking

We come now to a box construction that is less combinatorial in nature, called *locking*. Locking resembles lambda abstraction and 'quoting' (as in LISP), but also currying.

Let P be a function of type $\sigma \leftarrow \tau, \omega$, and let n denote the width of τ. Then we can 'lock' P, leaving in-type τ, by writing

$$\langle P \bullet ?_n \rangle : (\sigma \leftarrow \omega) \leftarrow \tau \quad .$$

The notation will shortly be explained, but first the meaning. This is a higher-order function; if fed with a τ-tuple, it produces a box (function) of type $\sigma \leftarrow \omega$. This box, when provided with an ω-tuple, yields that result that is produced by P when presented the τ-cum-ω-tuple in one go.

Now the notation. The $?_n$ is a *dummy*, or placeholder, for the τ-portion of the argument to P. Dummies are *only* allowed inside a $\langle \cdots \rangle$ form, called a locked expression, or for short a *lock*. *Within* that form, it is a box expression, formally typed

$$?_n : \vec{\alpha}^n \leftarrow \mathbb{1}$$

(in which the polymorphic type may be refined to some more specific type like $\tau \leftarrow \mathbb{1}$ for P above). So a dummy is formally a source. We put further

$$?_{m+n} \;=\; ?_m \| ?_n \;=\; ?_m \bullet ?_n \;=\; ?_m, ?_n \quad ,$$

and use $?$ for $?_1$. Furthermore, $?_0 = \mathrm{id}_0$, and so it may be eliminated as being the identity of each of the combinators.

The information in the subscript n of $?_n$ is crucial. For example, if we have $P : \alpha \leftarrow \beta, \gamma$, we can form three different locks:

$$\begin{aligned} &\langle P \rangle : && (\alpha \leftarrow \beta, \gamma) \leftarrow \mathbb{1} \quad , \\ &\langle P \bullet ? \rangle : && (\alpha \leftarrow \gamma) \leftarrow \beta \quad , \\ &\langle P \bullet ?_2 \rangle : && (\alpha \leftarrow \mathbb{1}) \leftarrow \beta, \gamma \quad . \end{aligned}$$

The *scope* of a dummy is the body of the locked expression in which it occurs, with the exclusion of other locks therein contained. Within its scope, all the laws that apply to sources may be applied to a dummy, such as (L16), giving us

$$?\oplus \;=\; \oplus \bullet ? \quad ,$$

or (L10), giving

$$? \bullet ?\|P \;=\; (? \bullet ?)\|P \;=\; ?\|P \bullet ? \quad .$$

There are important constraints, though. The first is an injunction against swapping dummies. In particular, dummies must not trade places by the rules for a mirrored operator; for example,

$$F \;=\; \langle ?\oplus? \rangle$$

has in general quite another meaning than

$$G \;=\; \langle ?\widetilde{\oplus}? \rangle \quad .$$

If $\oplus$ has type $\alpha \leftarrow \alpha{,}\beta$, then F would have type

$$(\alpha \leftarrow \mathbb{1}) \leftarrow \alpha{,}\beta \quad ,$$

whereas G would be typed

$$(\alpha \leftarrow \mathbb{1}) \leftarrow \beta{,}\alpha \quad .$$

The (L*)-laws given earlier are safe, however, *even if they appear to swap*. An example is provided by (L11), which can be instantiated to give

$$?_n \bullet P \;=\; \mathrm{id}_n || P \bullet ?_n \quad .$$

This allows us to shift dummies around, and it is valid even if P contains dummies. The explanation is that the dummy swap (if any) is only 'optically' present in the linearised box expression, and does not correspond to a swap in the topology of the network. For the human ease of application of rule (L19) given below, it is nevertheless helpful never to swap dummies, whether 'safe' or not.

A similar constraint must be exercised if algebraic properties of boxes are given, not only symmetry, but also having a zero, etc. For example, if for a given function P and given source C the property is known that

$$P \bullet A \;=\; C$$

for all sources A of in-width 1, this must not be used to simplify $P \bullet ?$ to C. A valid identity in this case is

$$P \bullet ? \;=\; C \bullet \bot \bullet ? \quad ,$$

in which the dummy, although 'sunk', is still visible. Like money, dummies may not be duplicated or embezzled. The (L*)-laws given earlier are also safe in this respect, with the exception of (L15) if the duplicated component contains dummies.

The following law allows moving a function across the boundary of a lock.

(L19) If P and R are dummy-free, and $\mathrm{id}_m \mathrel{\bullet\!\!\!\!=} R \mathrel{\bullet\!\!\!\!=} \mathrm{id}_n$, then

$$\langle P \bullet ?_m \bullet Q \rangle \bullet R \;=\; \langle P \bullet R \bullet ?_n \bullet Q \rangle \quad .$$

'Dummy-free' refers to dummies bound to the lock at this level; dummies of locked expressions contained within P or Q do not count. By combining this repeatedly with (L11), a bunch of functions can be moved across in one go. So as not to burden the exposition with excessive notation, here only the version with three functions is given, but the general pattern should be clear enough:

If all Pi and Ri are dummy-free, and $\mathrm{id}_{mi} \mathrel{\bullet\!\!\!\!=} Ri \mathrel{\bullet\!\!\!\!=} \mathrm{id}_{ni}$, then

$$\langle P0 \bullet ?_{m0} \bullet P1 \bullet ?_{m1} \bullet P2 \bullet ?_{m2} \bullet Q \rangle \bullet R0 || R1 || R2$$

$$=$$

$$\langle P0 \bullet R0 \bullet ?_{n0} \bullet P1 \bullet R1 \bullet ?_{n1} \bullet P2 \bullet R2 \bullet ?_{n2} \bullet Q \rangle \quad .$$

By supplying a source of the appropriate width, the body of a locked expression can be made dummy-free. If $P{:}\sigma \leftarrow \tau$ is dummy-free, then $\langle P \rangle$ has type $(\sigma \leftarrow \tau) \leftarrow \mathbb{1}$, so it is still a higher-order function. Something further is needed to set the locked function free.

15. Unlocking

Unlocking resembles application (or LISP eval), but also serves for uncurrying (when used in a left section). In this last aspect it is the counterpart of locking.

For each pair of naturals m and n we define a generic operator

$$@_{m,n}\colon \sigma \leftarrow (\sigma \leftarrow \omega), \omega \quad ,$$

in which the width of σ is m, and that of ω is n. The semantics are apparent from the type. Note that the in-width of the operator is $n+1$.

Let Q be a function of type $(\sigma \leftarrow \omega) \leftarrow \tau$, and let m and n as before denote the widths of σ and ω. Then we can 'unlock' Q by writing

$$Q\,@_{m,n}\colon \sigma \leftarrow \tau, \omega \quad .$$

The information in the subscripts m and n of $@$ is also contained in the type of its left operand. If this information is known from the context, the subscripts may be dropped. The notation $Q\,@$ implies then that Q has out-width 1, its out-type being some function type. Note, however, that more information is needed here than just the in- and out-width of Q, in particular the in-width of its out-type. By repeated unlocking, this need to know the widths can go arbitrarily deep into the type.

The major relationship linking unlocking with locking is given by:

(L20) If P is dummy-free, and $P \leqslant \mathrm{id}_n$, then

$$\langle P \bullet ?_n \rangle @ \;=\; P \quad .$$

We also have, so to speak, the converse of (L20), which allows us to express any higher-order function explicitly as a lock:

(L21) If Q has a function type as out-type, and in-width n, then

$$Q \;=\; \langle Q\,@ \bullet ?_n \rangle \quad .$$

16. Discussion

The objectives that had been set out at the start seem to have been achieved: the system is indeed centred around composition, is not opposed to typing, and has a relatively fair-handed treatment of, for example, both operand positions of an operator. The system comes with a rich (perhaps *too* rich) set of laws. If we use the names for values of the value world to name the corresponding sources in the function world, we get 'apposition' and 'lifting' for free, without the earlier semantic ambiguities.

A price is paid for all this—it is seldom that something really comes for free. In this case the price consists of conditions on most of the laws that are boring to verify.

The interest, if any, of this system is probably not in its theoretical properties. It has been constructed with a practical purpose in mind. The final judgment, therefore, must be how well it stands up in actual use. In particular, the question is if the additional convenience of the improved manipulability outweighs the inconvenience of the application conditions. As of now, the system has not yet been put to demanding tests. Some simple tests have been passed, but a lot more is needed before it can go into beta-test. The version presented here is not the first version; almost all notations, and several of the definitions, have undergone minor or sometimes dramatic changes since the inception of this line of research, and some combinators included at some time have been abandoned. In each case, the guiding principle has been to increase manipulability.

Several shortcomings are known that have not been pointed out. Some cherished notations of the formalism that served as the starting point, and also as a point of reference, do not fit in well. Most notable are the reduce notation, $\oplus/$, and the map notation, $f*$. The problem is that the operators $/$ and $*$ are not composed with, but applied to the (functional) operands. The notations introduced in this paper would require writing these as $\langle\oplus\rangle/$ and $\langle f\rangle*$, which is unacceptable. It is possible to introduce a special exemption here, which is kludgy, or to generalise this in some way, which unfortunately takes away some of the present simplicity of the system. Although in many cases less administrative overhead is needed than with any of several alternatives tried, and most of the time not even the surrogate variables provided by the dummies of a lock are needed, there are some examples where dummy variables do the job noticeably better than the anonymous ?-dummies used here. (A 'proof by picture' assigns essentially a name to the anonymous data, if only in the form of a pair of positions connected by a line on paper.)

Under the most liberal (semantic) definition of type-correctness, it is not decidable if an expression formed in this system can be typed. In each decidable typing discipline, some semantically unproblematic expressions are untypable. Without some restriction (as by a suitable typing), it is also undecidable if expressions are equivalent, and so there cannot be some canonical normal form. This is just as in the lambda calculus. It seems that next to locking and unlocking, also at least some duplicating Λ_n-function, possibly disguised as (id||id), is a necessary ingredient for the undecidability results.

Acknowledgements

The work reported here could not have been done without the inspiration provided by the unrelenting criticism of Richard Bird, but also, and much more so, by the elegant way in which he forged some of my earlier stumbling approaches into fine tools. It should be clear, however, that he is entirely without blame concerning the combinatorial tricks perpetrated here. Further inspiration came from many people, including Jeroen Fokker, Maarten Fokkinga, Netty van Gasteren, Johan Jeuring, Doaitse Swierstra, Nico Verwer, Jaap van der Woude, and Hans Zantema. The supportive context provided by the STOP project, sponsored by the Dutch National Facility for Informatics (NFI) and the Netherlands organization for scientific research (NWO), is gratefully acknowledged.

References

1. R. S. Bird. An introduction to the Theory of Lists. *Logic of Programming and Calculi of Discrete Design* (M. Broy, ed.) 5-42. NATO ASI Series Vol. F36, 1987. Springer-Verlag.
2. R. S. Bird. *Lectures on Constructive Functional Programming.* Lecture notes, International Summer School on Constructive Methods in Computing Science, Marktoberdorff, 1988.
3. E.W. Dijkstra, W. H. J. Feijen. *A Method of Programming.* Addison-Wesley, 1988.
4. L. Meertens. Algorithmics—Towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science,* CWI Monographs Vol. 1 (J.W. de Bakker, M. Hazewinkel and J.K. Lenstra, eds.) 289 – 334, North-Holland, 1986.